From C Programs to the Configure-Execute Model

João M. P. Cardoso[*]
FCT/University of Algarve,
Campus de Gambelas,
8000-117 Faro, Portugal
Email: jmpc@acm.org

Markus Weinhardt
PACT XPP Technologies AG
Muthmannstrasse 1,
80939 Munich, Germany
Email: mw@pactcorp.com

Abstract

The emergence of run-time reconfigurable architectures makes feasible the configure-execute paradigm. Compilation of behavioral descriptions (in, e.g., C, Java, etc.), apart from mapping the computational structures onto the available resources on the device, must split the program in temporal sections if it needs more resources than physically available. In addition, since the execution of the computational structures in a configuration needs at least two stages (i.e., configuring and computing), it is important to split the program such that the reconfiguration overheads are minimized, taking advantage of the overlapping of the execution stages on different configurations. This paper presents mapping techniques to cope with those features. The techniques are being researched in the context of a C compiler for the eXtreme Processing Platform (XPP). Temporal partitioning is applied to furnish a set of configurations that reduces the reconfiguration overhead and thus may lead to performance gains. We also show that when applications include a sequence of loops, the use of several configurations may be more beneficial than the mapping of the entire application onto a single configuration. Preliminary results for a number of benchmarks strongly confirm the approach.

1 Introduction

Run-Time Reconfigurable (RTR) architectures promise to be efficient solutions when flexibility, high-performance, and low power consumption are required features. Since Field-Programmable Gate Arrays (FPGAs) require long design cycles with low level hardware efforts and the fine-granularity (bit level) used does not match many of today's applications, new reconfigurable processing units (RPUs) are being introduced [1]. However, compilation research efforts are still required in order to facilitate programming on those architectures. Specifically, the mapping of large programs must be automatically performed.

Moreover, those architectures deal with complex programs by executing sequences of configurations. Those configurations are obtained by splitting the computational structures onto temporal partitions. Since for executing each partition at least the stages configure and execute are required, the model is called configure-execute. In order to enhance performance, overlapping of different stages should be considered. We have researched methods to exploit those features. The methods take advantage of the configure-execute paradigm while trying to reduce the overall execution time by hiding some of the reconfiguration time. This paper includes some of our research efforts, which are related to the work that we have been carrying on for compiling C programs to the XPP architecture [2], a data-driven, coarse-grained, 2D array parallel structure.

This paper is organized as follows. The next section introduces briefly the XPP technology. Section 3 explains the configure-execute paradigm when using the XPP. Section 4 describes our approach to automatically output a set of configurations from a C program attempting to minimize the overall execution time. Section 5 shows some experimental results and section 6 describes the related work. Finally, section 7 concludes the paper and summarizes ongoing and planned future work.

2 The XPP technology

The XPP architecture is based on a 2D array of coarse-grained, adaptive processing elements (PEs), internal memories, and interconnection resources. The XPP has some similarities with other coarse-grained reconfigurable architectures, which have been especially designed for stream-based applications (*e.g.,* Function Processor [3], KressArray [4], and RaPiD [5]). XPP's main distinguishing features are its sophisticated configuration mechanisms.

Fig. 1 (taken from [6]) shows the structure of a simple XPP. It contains a square of PEs in the center and one column of independent internal memories on each side. There are two I/O units which can either be configured as ports for streaming data or as interfaces for external RAM. The PEs perform common arithmetic and logical operations, comparisons, and special operations such as counters. In each configuration, a PE performs one dedicated operation. Each

[*]This work has been partially done during the year the author has spent with PACT XPP Technologies AG.

1530-1591/03 $17.00 © 2003 IEEE

thick line in the figure represents several segmented busses which can be configured to connect PEs. The array is coupled with a *Configuration Manager* (CM) responsible for the runtime management of configurations and for downloading configuration data from external memory into the configurable resources of the array. Besides a finite state machine, the CM has cache memory for accommodating multiple configurations.

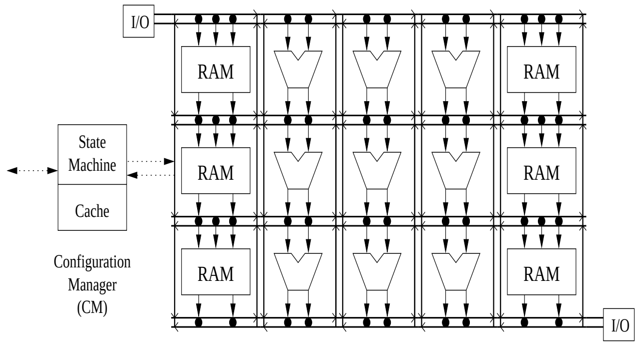


Figure 1: Simplified structure of an XPP array.

The interconnection resources on the array consist of two independent sets of busses: data and event busses. Data busses have a uniform bit width specific to the device type, while event busses carry one-bit control information. The busses include a ready/acknowledge protocol to synchronize the data and events processed by the PEs. Pipelining registers can be switched on or off in the bus segments. A PE operation is performed as soon as all necessary input values are available and the previous result has been consumed. Similarly, a value produced by a PE is forwarded as soon as all the PE's receivers have consumed the previous value. Note that an XPP is a synchronous machine operating at a fixed clock frequency.

Events transmit state information which can be used to control PEs' execution or memory accesses. The PEs can perform operations such as MERGE, DEMUX, and MUX. MERGE uses an event to select one of two operands. MUX has a similar functionality, but discards the data on the input not selected. Finally, DEMUX forwards its input data to the selected output. DEMUX can also be used to selectively discard data. Some PE operations generate events depending on results or exceptions. A counter, *e.g.*, generates a special event only after it has terminated.

Every configurable object (*e.g.*, PE, memory) locally stores its current configuration state, *i.e.*, if it is part of a configuration or not (states "configured" or "free"). If a configuration is requested, its configuration data is first copied to the internal cache. Then the CM downloads the configuration onto the array, synchronizing with the configurable objects. Once an object is configured, it changes its state to "configured". This prevents the CM from reconfiguring an object which is still used by another configuration. Objects are released (*i.e.*, changed to state "free") by events on a special input. These events are automatically broadcast along all connected PEs such that the entire configuration is removed. Events on release inputs can be explicitly generated by the CM or created in the array itself by a PE.

The XPP is supported by a complete development tool suite (XDS) [2], consisting of a place and router (`xmap`), a simulator (`xsim`), and a visualization tool (`xvis`). The architecture can be programmed using the proprietary *Native Mapping Language* (NML). In NML, configurations consist of modules, which are specified in a structural manner: PEs are explicitly allocated, optionally placed, and their connections specified. Additionally, NML includes statements to specify configuration handling. `xmap` compiles the NML files, places and routes the objects, and generates XPP binary files. Those binaries can either be simulated cycle by cycle or directly executed on an XPP device. A high-level compiler has been added to XDS and permits to map C programs onto the XPP [6].

3 The configure-execute paradigm

The execution of a configuration in the XPP requires three stages: fetching, configuring, and computing. Fetching is related to the load of the configuration data to the CM cache. Configuring is related to the download of the configuration data from the cache onto the array structures. Computing is related to the execution on the array. We include in the computing stage the release of the used resources after completion of execution, whenever such release is required. This self-release of the resources makes possible to execute an application consisting of several configurations without any external control. When a computation is finished, the PE detecting the end (*e.g.*, a counter) can generate the event to be broadcast or can send an event to the CM requesting the next configuration. The execution stages can be done in parallel for different configurations (*e.g.*, while doing the configuring or computing stages of a configuration the fetch of another configuration can be performed). Because of its coarse-grain nature, an XPP can be configured more rapidly than FPGAs. Since only the configuration of those array objects actually used is necessary, the fetching and configuring times depend on each configuration.

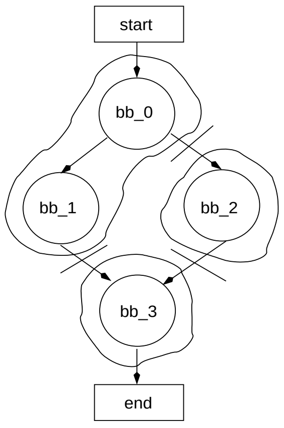
A program too large to fit in an XPP can be handled by splitting it in several parts (configurations) such that each one is mappable [6]. Although that approach enables the mapping of large computational structures onto the available resources, other goals must be considered. An automatic approach must consider a judicious selection of a set of configurations, such that the overall execution time of the application is minimized and it is sucessfully mapped onto the XPP resources. The costs to load into the cache, to configure and to compute each configuration with the XPP must be taken into account.

The minimization of the overall execution time can be mainly achieved by the following issues: (1) reduction of each partition's complexity can reduce the interconnection delays (long interconnections may pass through registers requiring more clock cycle); (2) reduction of the number of references, in the section of the program related to each partition, using the same resource, by distributing the overall references among partitions, might lead to performance gains, as well. This happens with the statements presented in the program referring the same array; (3) reduction of the overall configuration overhead by overlapping fetching, configuring and computing of distinct partitions.

Example 1: Consider the C example `max_avg` shown in Fig. 2(a), where configuration boundaries are represented by annotations. They define three regions of code (see Fig. 2(b)). Combining only the most frequently taken conditional paths in the same partition can reduce the total execution time by substantially reducing the reconfiguration time (since the partitions for the other paths are not configured when they are not taken). In Fig. 2(b) if the path bb_0 and bb_1 has been identified as the most frequently executed, this path can be in the same partition. In that case, the configuration related to bb_2 will only be requested when this branch is taken.

```
// max_avg example
...                            // bb_0
if(op==1) {                    // bb_0
    // average kernel          // bb_1
    sum = 0;                   // bb_1
    for(i=0;i<N; i++) {        // bb_1
        sum+=x[i];             // bb_1
    }                          // bb_1
    average = sum/N;           // bb_1
    // configuration boundary
} else {
    // configuration boundary
    // max kernel              // bb_2
    max = 0;                   // bb_2
    for(i=0;i<N; i++) {        // bb_2
        if(x[i] > max)         // bb_2
            max = x[i];        // bb_2
    }                          // bb_2
    // configuration boundary
}
...                            // bb_3
```

(a)



(b)

Figure 2: Example with two conditionally executed kernels: (a) C code; (b) CFG. Lines crossing edges represent the "configuration boundaries" annotated in the code. Bubbles containing basic blocks represent the regions to be implemented in different partitions/configurations.

Example 2: The NML code listed in Fig. 3 represents an application with two configurations. One can see in the APPLICATION section the specification of the reconfiguration control flow, which will be executed by the CM. The example instructs the CM to prefetch the two configurations. Note, however, that the configuration MOD0 is downloaded onto the array as soon as its configuration data is on the CM cache (concurrently, the configuration data of MOD1 is downloaded onto the CM cache). The example assumes that each configuration (MOD0 and MOD1) self-releases its resources after completion of computing. NML includes statements to conditionally request configurations, which are used in cases where the configuration to be requested depends on a value of an event generated in the array.

```
XPP(...) // header identifying the parameters of the used XPP
MODULE MOD0 {   // structure of configuration MOD0}
MODULE MOD1 {   // structure of configuration MOD1}
APPLICATION example {
  START(pre_fetch, c_MOD0) // this is the start command
  CONFIG pre_fetch {
    PREFETCH(c_MOD0) // request the prefetch of MOD0
    PREFETCH(c_MOD1) // request the prefetch of MOD1
  }
  CONFIG c_MOD0 {
    CONF_MODULE(MOD0) // start configuring MOD0
    REQUEST(c_MOD1) // request c_MOD1
  }
  CONFIG c_MOD1 {
    CONF_MODULE(MOD1) // start configuring MOD1
  }
}
```

Figure 3: Example of NML code, including the description of the reconfiguration control flow.

4 Generation of configurations

As we can see by the aforementioned features, the XPP is specially tailored to support the configure-execute paradigm. But, from the point of view of the compilation, which methods can be really employed to automatically generate the configurations that will run on the platform? Furthermore, how temporal partitioning can be performed in order to reduce the overall execution latency (*e.g.*, by hiding some reconfiguration overheads)? This section presents our proposed approach, which is being tested in the XPP-VC compiler [6]. The compiler is based on the SUIF compiler framework [7]. Fig. 4 shows the XPP-VC compilation flow. The `MODGen` task of the compiler transforms a sequential C description in a data- and event-driven control/dataflow graph (CDFG), which can be directly mapped to the structures of the XPP array. It uses a technique based on the *Pipeline Vectorization* method developed for reconfigurable architectures [8] to vectorize inner program FOR-loops, such that loop iterations are executed in a pipelined, parallel fashion. A C program can be manually splited in several sections, each one corresponding to a different configuration, by using annotations. Otherwise, one can use automatic temporal partitioning in order to furnish mappable partitions [6]. The compiler generates both the NML representation of each partition and the NML section specifying the control flow of configurations. Such control flow is orchestrated by the CM of the XPP during runtime, as has been already explained. Finally, xmap is used to place and route each configuration structure, to generate the configuration data, and the binaries to program the CM.

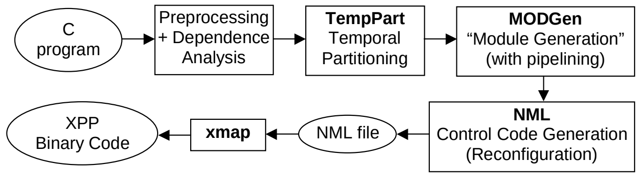

C program
Preprocessing + Dependence Analysis
TempPart Temporal Partitioning
MODGen "Module Generation" (with pipelining)
NML Control Code Generation (Reconfiguration)
NML file
xmap
XPP Binary Code

Figure 4: XPP-VC compilation flow.

Our method starts by constructing, from the SUIF representation of the C program, an extended Hierarchical Task Graph[1], HTG+. This graph has two types of nodes: (1) *behavioral nodes* representing program statements; (2) *array nodes* representing each array variable in the program. Fig. 5 shows the top level of the HTG+ for an implementation of the Discrete Cosine Transform (DCT) based on matrix multiplications. Type (1) nodes have three distinct subtypes: (a) *block nodes* representing basic blocks; (b) *compound nodes* representing `if-then-else` structures; (c) *loop nodes* representing loops. *Loop* and *compound nodes* explicitly embody sub-HTG+s. Edges in the HTG+ represent data communication between two nodes or just enforce execution's precedence.

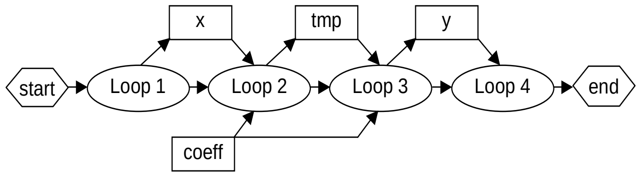


Figure 5: HTG+ (top level) for the DCT example. Circles and boxes represent *behavioral* and *array nodes*, respectively. Data is read from an input port (Loop1) and written to an output port (Loop4).

Estimation of the resources and execution latency on the XPP for implementing each *behavioral node* is then performed. The HTG+ nodes are labeled with the following information: (1) *block and compound nodes*: number of ALUs and REGs; (2) *loop nodes*: number of iterations (if unbound, profiling should be used), and number of ALUs and REGs; (3) *array nodes*: the size of the array, type of the elements, and, when they do exist, the initialization values. Each edge between two *behavioral nodes* is labeled with the number of data words that might be transferred between them. Each edge between an *array node* and a *behavioral node* is labeled with the number of load and store references in the source code represented by the *behavioral node* to that particular array. The estimated number of times that each load and store reference will be executed is also collected. This is used to evaluate the impact of mapping several computational structures in the same partition, since the use of the same array by different *behavioral nodes*, might increase the execution latency and the number of required resources[2].

The approach computes the number of XPP resources needed and the computing latency by each configuration implementing a single or a set of behavioral nodes. Based on the estimation of the used resources, the approach estimates the number of clock cycles to fetch and configure each partition (computed using the estimation of the needed resources directly from the SUIF representation or with the number of edges, ALUs, REGs, and pre-defined values existent in each CDFG generated by MODGen).

After those steps, the algorithm starts with one partition/configuration per node on the top of the HTG+. Then, it attempts to merge adjacent partitions until no performance gains are achieved taking also into account the size constraint. Each considered merge is only effectively performed if it results in a reduction of the estimated overall execution latency. The estimation takes into account the concurrency of the fetching, configuring, and computing stages for different partitions. Since the XPP is partially configurable, we use a conservative assumption that when a configuration (c1) is computing on the array and another one (c2) is being downloaded onto the array by the CM, the only configuration words that can be configured in parallel are those related to the number of resources of c2 exceeding the number of resources used by c1.

The algorithm only considers partitioning sub-HTG+s if a top level node cannot be mapped. Since the time to fetch and configure may have a significant impact, it is in most cases preferable to reuse a configuration as long as possible in order to reduce the reconfiguration time overhead. Thus, loops in the program are always good candidates to be entirely implemented by a single configuration. The strategy only exploits loop partitioning if an entire loop cannot be mapped to the XPP or contains sequences of loops in its body. When those cases occur, the algorithm is applied to the body of the loop and the *loop dissevering* technique [6] is used.

Each partition must currently define, on the control flow graph (CFG) of the program, regions of code with all entries to the same instruction and possibly multiple exists. For each exit point existent in each partition there will be an event connected to one of the CM ports available in the XPP (the CM can check if an event is generated and can proceed with different configurations based on the value of the event).

To furnish feasible temporal partitions (*i.e.*, partitions that can be truly mapped onto the XPP) the approach ulti-

[1]The model has been chosen, because it can also be used to explicitly represent loop and task level parallelism.

[2]*E.g.*, twice the number of references to the same RAM leads to more than twice the number of objects required on XPP, and delays each access because of the objects needed to MERGE and DEMUX data and address values.

mately uses checks with MODGen and xmap before furnishing the set of partitions. Since it might not be tolerable to check each performed merge, the algorithm only checks the final merges with xmap. Everytime the estimated resources using MODGen exceed the size constraint, there is no need to call xmap. The approach is explained in [6] and it uses three levels of checks and performs merge steps decreasing the size constraint until a feasible set is found.

5 Experimental results

Tab. 1 shows some results obtained when compiling a set of benchmarks with the automatic temporal partitioning scheme presented in [6] (1) and with the new scheme proposed in this paper (2). Note that none of the examples shown was specially coded to exploit more efficiently the architectural features of the XPP (*e.g.*, partitioning and distribution of arrays among the internal memories) and thus the results can be further improved. We use an hypothetic XPP array with 16×16 PEs. Columns #loC, #LPs, #cf, and #PEs, represent the number of lines of C code, number of loops, number of configurations, number of used PEs (it is shown the maximum number of PEs of the largest configuration and the total number of PEs virtually needed), respectively. On column Exec we show, for each example, the number of clock cycles (150 MHz can be considered as a typical clock frequency) of the overall execution latency (taken into account the setup, fetching, configuring, data communication and computing stages).

Table 1: Results obtained with the proposed approach.

Benchs	#loC	#LPs	scheme	#cf	#PEs	Exec (#ccs)
DCT1 (a)	60	8	(1)	1	85/85	10,406
DCT1 (b)			(2)	4	36/83	8,937
DCT1 (c)			(1)	1	213/213	16,400
DCT1 (d)			(2)	4	103/217	9,596
Chen (a)	169	5	(1)	1	286/286	20,424
Chen (b)			(2)	5	180/398	11,624
Lee (a)	229	4	(1)	1	288/288	20,014
Lee (b)			(2)	4	220/458	11,768
Haar (a)	55	8	(1)	1	127/127	82,200
Haar (b)			(2)	4	57/172	76,406
Life (a)	118	10	(1)	4	144/304	2,516,652
Life (b)			(2)	6	123/416	1,282,540

DCT1 is an 8×8 DCT implementation which is based on two matrix multiplications (see the top level of its HTG+ in Fig. 5). Splitting the program improves the overall latency of DCT1 by 14% requiring 58% less PEs ((a) vs (b)). When considering each of the innermost loops in the matrix multiplications fully unrolled, splitting the program improves the overall latency of DCT1 by 41% requiring 52% less PEs ((c) vs (d)). **Chen** and **Lee** are pointer-free versions of two other DCT implementations. Our proposed scheme furnishes improved versions: 43% in performance requiring 12% less PEs for **Chen** ((a) vs (b)) and 41% in performance needing 24% less PEs for **Lee** ((a) vs (b)). **Haar** is an implementation of the forward 2D Haar wavelet transform. An input image of 64×64 is used. An increase in performance of 7% and a reduction of 55% in size are achieved with the solution obtained with our scheme ((a) vs (b)). **Life** refers to the Conway's Game of Life included in the RawBench suite (using a 32×32 grid and 8 iterations). An increase in performance of 49% and a reduction of 15% in size is achieved when using 6 instead of 4 configurations ((a) vs (b)). The version with 6 configurations uses *loop dissevering* to partitioning the WHILE loop responsible for the iterations.

For each example, the compilation, from the source program to the generation of the binary configuration file, is performed in few seconds (using a Pentium III at 933 MHz with Linux).

The results show that using our temporal partitioning approach leads to sets of configurations that reduce the reconfiguration overhead and thus may lead to performance gains. The results strongly confirm that when applications include a sequence of loops, the use of several configurations may be more beneficial than the mapping of the entire application onto a single configuration. The achieved performance gains with solutions using several configurations are more evident when each kernel computes on small data sets and thus the reconfiguration overhead is significant. Although when larger data sets are used the performance gains might be insignificant, the size reduction on the number of needed resources justifies the use of the approach.

6 Related work

The work on compiling high-level descriptions onto reconfigurable logic has been the focus of many researchers since the first simple attempts (*e.g.*, [9]). Not so many compilers have considered the integration of temporal partitioning. Probably, the first compiler to include temporal partitioning was the dbC Napa compiler [10]. In that approach, each function defines a configuration and the only way to control the generation of configurations is to manually group program statements in functions. An automatic approach has been used in the Galadriel & Nenya compiler [11]. The initial program is transformed in an intermediate representation graph where temporal partitioning is performed. Other authors compile to architectures, which explicitly support hardware virtualization (*e.g.*, PipeRench [12]). However, they only support acyclic computations or computations where cyclic structures can be transformed to acyclic ones.

Most of the current approaches attempt to achieve a minimum number of configurations (*e.g.*, [13]). Those schemes only consider another partition after the current one has filled the available resources and are insensible to the optimization that must be applied to reduce the overall execution by overlapping the fetching, configuring and computing steps. One of the first attempts to reduce the configuration overhead in the context of temporal partitioning has been presented in [14]. They attempt to overlap configuring with computing of adjoining partitions. The approach uses the simple model of splitting the available FPGA resources into two parts and mainly performing temporal partitioning using half of the total available area as the size constraint.

In the context of scheduling configuration kernels, an approach targeting the MorphoSys architecture has been presented in [15]. They attempt to minimize the reconfiguration overhead and to maximize data reuse. The approach uses an exploration of the search space with some pruning features. However, no automatic temporal partitioning is considered and they assume the presence of each kernel's configuration data.

When performing automatic temporal partitioning of high-level programs, the work presented in this paper is, to the best of our knowledge, the first one that attempts to achieve the minimization of the overall execution time by considering the overlapping of the different stages to execute each configuration, and the overheads that might occur when computation structures are grouped into the same configuration.

7 Conclusions

This paper describes preliminary results obtained with a novel scheme for mapping software programs onto a reconfigurable computing platform. The approach, supported by temporal partitioning, enables the mapping of complex programs, and attempts to furnish implementations with maximum performance by hiding some of the reconfiguration time. The scheme takes into account the size and latency overheads, when computational structures accessing the same resource (*e.g.*, memories) are merged onto the same configuration. The obtained results prove that temporal partitioning must be sensible to those overheads and to the hiding of reconfiguration times, while deciding on the best set of configurations to implement a certain application. A judicious selection of configurations might reduce the overall execution time, by furnishing solutions that overlap the execution stages needed for each configuration, as is demonstrated by the results achieved.

Considering the configure-execute model, the results show that in many cases a smaller array can be used without sacrificing performance. We expect to continue the evaluation with a set of applications with a larger number of kernels. Ongoing work focuses on tuning the estimation steps and on improving the configuration data generated. Finally, an extension of the approach to cope with the automatic exploitation of *loop distribution* is planned.

References

[1] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," In *Proc. Design, Automation and Test in Europe (DATE'01)*, 2001, pp. 642-649.

[2] PACT XPP Technologies AG, Munich, Germany, "The XPP White Paper," Release 2.0, June 2001. http://www.pactcorp.com

[3] J. Vasell, and J. Vasell. "The Function Processor: A Data-Driven Processor Array for Irregular Computations," in *Future Generation Computer Systems*, 8(4), 1992, pp. 321-335.

[4] R. W. Hartenstein, R. Kress, and H. Reining, "A Dynamically Reconfigurable Wavefront Array Architecture for Evaluation of Expressions," in *Proc. Int'l Conf. on Application-Specific Array Processors (ASAP'94)*, IEEE Computer Society Press, 1994.

[5] D. C. Cronquist, et al., "Architecture Design of Reconfigurable Pipelined Datapaths," In *20th Anniversary Conf. on Advanced Research in VLSI*, Atlanta, GA, USA, March 1999, pp. 23-40.

[6] J. M. P. Cardoso, and M. Weinhardt, "XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture," in *Proc. 12th Int'l Conf. on Field Programmable Logic and Applications (FPL'02)*, LNCS 2438, Springer-Verlag, 2002, pp. 864-874.

[7] SUIF Compiler system, "The Stanford SUIF Compiler Group," http://suif.stanford.edu

[8] M. Weinhardt, and W. Luk, "Pipeline Vectorization," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Feb. 2001, pp. 234-248.

[9] I. Page, and W. Luk, "Compiling occam into FPGAs," In *FPGAs*, Will Moore and Wayne Luk, eds., Abingdon EE&CS Books, Abingdon, England, UK, 1991, pp. 271-283.

[10] M. Gokhale, and A. Marks, "Automatic Synthesis of Parallel Programs Targeted to Dynamically Reconfigurable Logic Array," in *Proc. 5th Int'l Workshop on Field Programmable Logic and Applications (FPL'95)*, LNCS, Springer-Verlag, 1995, pp. 399-408.

[11] J. M. P. Cardoso, and H. C. Neto, "Macro-Based Hardware Compilation of Java Bytecodes into a Dynamic Reconfigurable Computing System," in *Proc. IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*, IEEE Computer Society Press, 1999, pp. 2-11.

[12] S. C. Goldstein, et al., "PipeRench: A Reconfigurable Architecture and Compiler," in *IEEE Computer*, Vol. 33, No. 4, April 2000.

[13] I. Ouaiss, et al., "An Integrated Partioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," in *Proc. 5th Reconfigurable Architectures Workshop (RAW'98)*, Orlando, Florida, USA, March 30, 1998, pp. 31-36.

[14] S. Ganesan, and R. Vemuri, "An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement," in *Proc. Design, Automation & Test in Europe (DATE'00)*, Paris, France, March 27-30, 2000, pp. 320-325.

[15] R. Maestre, et al., "A Framework for Reconfigurable Computing: Task Scheduling and Context Management," in *IEEE Transactions on VLSI Systems*, Vol. 9, No. 6, Dec. 2001, pp. 858-873.